Algoritmos y Estructura de Datos: Examen 1 (Solución)

Grados Ing. Inf. y Mat. Inf. Octubre 2011

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Este examen consta de **6 preguntas** en **8 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **90 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **2 de noviembre de 2011** junto con las soluciones. La fecha de la revisión de este examen es el **4 de noviembre de 2011**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.
- En las preguntas con varias opciones, **sólo hay una respuesta válida por pregunta**. En este caso toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada y toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma menos uno. Es decir, una respuesta incorrecta en una pregunta de un punto con cuatro alternativas resta $\frac{1}{3}$ de punto.

(1 punto) 1. Dada la implementación de listas indexadas `IndexList` mediante vectores `ArrayIndexList`. **Se pide:** Indicar de entre las siguientes posibilidades la que describe correctamente la complejidad del método de inserción `add` para el caso peor. Se asume que n es el tamaño de la lista:

(a) $O(1)$ porque añadir un elemento a un vector de Java siempre tiene complejidad constante.

(b) $O(n)$ porque en el caso peor hay que desplazar todos los elementos del vector a la izquierda para hacer hueco al nuevo elemento.

(c) $O(n)$ porque en el caso peor hay que desplazar todos los elementos del vector a la derecha para hacer hueco al nuevo elemento. ✔

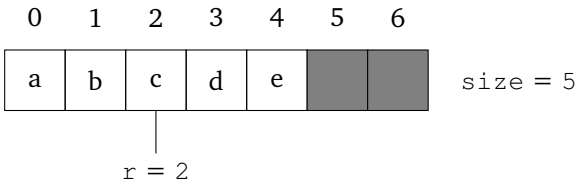
(d) $O(n)$ porque en el caso peor siempre hay que copiar los elementos del vector a un nuevo vector más grande y luego desplazar los elementos.

(1 punto) 2. Dado el siguiente fragmento incompleto de la clase `ArrayIndexList<E>`:

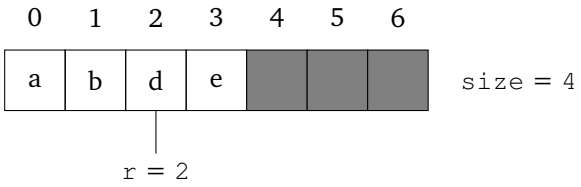
```
public class ArrayIndexList<E> implements IndexList<E> {
  private E[] A;
  private int capacity = 6;
  private int size;
  ...
  /** Checks whether the given index is in the range [0, n-1] */
  protected void checkIndex(int r, int n)
    throws IndexOutOfBoundsException {
    if (r < 0 || r >= n)
```

```
        throw new IndexOutOfBoundsException("Illegal index: " + r);
    }
    ...
    public E remove(int r) throws IndexOutOfBoundsException {
      checkIndex(r, size());
      E temp = A[r];

      /** COMPLETAR **/

      return temp;
    }
  }
```

Se pide: Completar el método `remove` con el código que elimina el elemento `A[r]` del vector. Por ejemplo, cuando el vector contiene los siguientes 5 elementos y se invoca `remove(2)`:

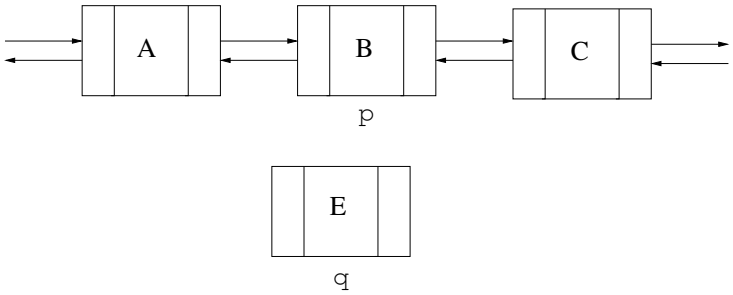


El vector debe quedar de la siguiente manera:




```
for (int i=r; i<size-1; i++)
  A[i] = A[i+1];
size--;
```

(2 puntos) 3. La siguiente figura muestra un fragmento de una lista doblemente enlazada de objetos de clase `DNode<E>` (código de la clase disponible en el Apéndice A.2). Asumimos que las variables `p` y `q` almacenan los objetos nodo indicados.



Se pide: Escribir las líneas de código que insertan el nodo `q` inmediatamente después del nodo `p` en la cadena doblemente enlazada, de forma que la lista contenga la secuencia de elementos A, B, E, C.

```
q.setNext(p.getNext());
q.setPrev(p);
p.getNext().setPrev(q);
p.setNext(q);
```

(1 punto) 4. Se tiene una aplicación de gestión de pedidos que necesita almacenarlos en el orden inverso en que han sido realizados. Se decide usar un TAD lista para almacenar los pedidos. Cuando un cliente realiza un pedido p_1 y después otro pedido p_2 éstos se almacenan en la lista en orden inverso: p_2, p_1. Se dispone únicamente de las implementaciones `ArrayIndexList` y `NodePositionList`. **Se pide:** Razonar cuál de las dos implementaciones sería la idónea en términos de mejor complejidad de almacenamiento.

La idónea es `NodePositionList` pues su método `addFirst(p)` inserta el último pedido `p` al principio de la lista con complejidad $O(1)$ en el caso peor, mientras que el método `add(0,p)` de `ArrayIndexList` inserta el último pedido `p` al principio de la lista con complejidad $O(n)$ en el caso peor.

(2 puntos) 5. Se desea añadir el siguiente método al interfaz `List<E>` (disponible en el Apéndice A.3):

```
public void append(List<E> list)
```

El método debe «concatenar» la lista que se pasa como parámetro a la lista sobre la que se invoca el método. Dada la invocación `list1.append(list2)` el método debe añadir en orden de aparición los elementos del objeto `list2` al final del objeto `list1`. Por ejemplo, si `list1` contiene los elementos 1,3,4 y `list2` los elementos 7,8, después de la invocación `list1` contendrá los elementos 1,3,4,7,8. Obsérvese que `append` **no** lanza excepciones. **Se pide:** Escribir la implementación del método utilizando únicamente los métodos ofrecidos por el interfaz `List<E>`.

```
public void append(List<E> list) {
   for (int i = 0; i < list.size(); i++) addLast(list.get(i));
 }
```

(3 puntos) 6. **Se pide:** Implementar en Java el siguiente método:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2)
```

que inserta en una lista de posiciones dada `list` (interfaz `PositionList<E>` disponible en Apéndice A.4) el elemento `e1` en el nodo (`Position`) justamente anterior al que almacena el elemento `e2`, si este último está en la lista. Si `e2` no está en la lista el método deja la lista intacta.

Solución que inserta `e1` antes de la primera ocurrencia de `e2`:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2) {
  if (!list.isEmpty()) {
    Position<E> p = list.first();
    for (int n = 1; n < list.size() && !p.element().equals(e2); n++)
      p = list.next(p);
    // Aquí p es o la posición con e2 o la última posición.
    if (p.element().equals(e2)) list.addBefore(p,e1);
  }
}
```

Solución que inserta `e1` antes de todas las ocurrencias de `e2`:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2) {
  for (Position<E> p : list.positions())
    if (p.element().equals(e2)) list.addBefore(p,e1)
}
```

A. Código de apoyo

A.1. Interfaz `IndexList<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for array lists.
 * @author Roberto Tamassia, Michael Goodrich
 */
public interface IndexList<E> {

  /** Returns the number of elements in this list. */
  public int size();

  /** Returns whether the list is empty. */
  public boolean isEmpty();

  /** Inserts an element e to be at index i,
    * shifting all elements after this. */
  public void add(int i, E e) throws IndexOutOfBoundsException;

  /** Returns the element at index i, without removing it. */
  public E get(int i) throws IndexOutOfBoundsException;

  /** Removes and returns the element at index i, shifting the elements
    *  after this. */
  public E remove(int i) throws IndexOutOfBoundsException;

  /** Replaces the element at index i with e, returning the
    *  previous element at i. */
  public E set(int i, E e) throws IndexOutOfBoundsException;
}
```

A.2. Clase `DNode<E>`

```
package net.datastructures;
/**
 * A simple node class for a doubly-linked list.  Each DNode has a
 * reference to a stored element, a previous node, and a next node.
 *
 * @author Roberto Tamassia
 */
public class DNode<E> implements Position<E> {
  private DNode<E> prev, next;  // References to the nodes before and after
  private E element;    // Element stored in this position
  /** Constructor */
  public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {
    prev = newPrev;
    next = newNext;
    element = elem;
  }
  // Method from interface Position
  public E element() throws InvalidPositionException {
    if ((prev == null) && (next == null))
      throw new InvalidPositionException("Position␣is␣not␣in␣a␣list!");
    return element;
  }
  // Accessor methods
  public DNode<E> getNext() { return next; }
  public DNode<E> getPrev() { return prev; }
  // Update methods
  public void setNext(DNode<E> newNext) { next = newNext; }
  public void setPrev(DNode<E> newPrev) { prev = newPrev; }
  public void setElement(E newElement) { element = newElement; }
}
```

A.3. Interfaz `List<E>`

```
package listLibrary;
import java.util.Iterator;
import java.lang.IndexOutOfBoundsException;

public interface List<E> extends Iterable<E> {

  /* Tamaño de la lista */
  public int size();

  /* Indica si la lista está vacía */
  public boolean isEmpty();

  /* Devuelve el primer elemento de la lista.
   * Si la lista está vacía lanza EmptyListException. */
  public E first() throws EmptyListException;

  /* Devuelve el último elemento de la lista.
   * Si la lista está vacía lanza EmptyListException. */
  public E last() throws EmptyListException;

  /* Devuelve el elemento en la posición i.
   * Si la lista está vacía lanza EmptyListException.
   * Si no se cumple 0 <= i < size() lanza IndexOutOfBoundsException. */
  public E get(int i) throws EmptyListException, IndexOutOfBoundsException;

  /* Inserta el elemento al principio de la lista */
  public void addFirst(E e);

  /* Inserta el elemento al final de la lista */
  public void addLast(E e);

  /* Vacía la lista */
  public void reset();

  /* Devuelve el elemento en la posición i y lo reemplaza por el nuevo.
   * Si no se cumple 0 <= i < size() lanza IndexOutOfBoundsException */
  public E set(int i, E e) throws EmptyListException, IndexOutOfBoundsException;

  /* Devuelve el elemento en la posición i y lo borra de la lista.
   * Si no se cumple 0 <= i < size() lanza IndexOutOfBoundsException */
  public E remove(int i) throws EmptyListException, IndexOutOfBoundsException;

}
```

A.4. Interfaz `PositionList<E>`:

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for positional lists.
 * @author Roberto Tamassia, Michael Goodrich
 */
public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of elements in this list. */
  public int size();

  /** Returns whether the list is empty. */
  public boolean isEmpty();

  /** Returns the first node in the list. */
  public Position<E> first();

  /** Returns the last node in the list. */
  public Position<E> last();

  /** Returns the node after a given node in the list. */
  public Position<E> next(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the node before a given node in the list. */
  public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Inserts an element at the front of the list, returns new position. */
  public void addFirst(E e);

  /** Inserts and element at the back of the list, returns new position. */
  public void addLast(E e);

  /** Inserts an element after the given node in the list. */
  public void addAfter(Position<E> p, E e)
    throws InvalidPositionException;

  /** Inserts an element before the given node in the list. */
  public void addBefore(Position<E> p, E e)
    throws InvalidPositionException;

  /** Removes a node from the list, returning the element stored there. */
  public E remove(Position<E> p) throws InvalidPositionException;

  /** Replaces the element stored at the given node, returns old element. */
  public E set(Position<E> p, E e) throws InvalidPositionException;

  /** Returns an iterable collection of all the nodes in the list. */
  public Iterable<Position<E>> positions();

  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();
}
```